ECE 4750 Lab 2: Pipelined Processor

Basant Amro Khalil (bak94)

October 20, 2021

Section 1. Introduction

In this lab, we are comparing two pipelined processor implementations, one that uses stalling to handle data hazards, while the alternative design uses both stalling and bypassing. Pipelining involves splitting the datapath and control modules into multiple stages using buffer registers with the potential speedup proportional to the number of pipelined stages. It enables the overlapping of the execution of multiple transactions. As such, pipelined processors are in general able to have a faster cycle time over single-stage processors since the critical path is shorter, while still executing instructions on average approximately once per cycle. In this lab, we create two designs for a 5-stage pipelined processor that supports the TinyRV2 ISA. Unfortunately, pipelining introduces different types of data hazards that need to be resolved. The baseline design uses stalling to resolve data hazards, while the alternative design uses bypassing to improve the processor performance. The bypassing in the alternative design resolves data dependencies more efficiently, requiring either zero or one cycle to resolve data hazards. Bypassing works by using special bypass paths from the Execute stage(X), Memory stage (M), and the Write-Back stage (W) to the Decode stage (D). These bypass paths enable hazards created by data dependencies to be resolved as fast as in the same cycle, eliminating the need to stall for many instructions (of which load (lw) and store (sw) are exceptions, for instance). We expect an improvement in the transaction throughput for the alternative bypassed design compared to the baseline design (without bypassing). In this lab, we implemented both designs, verified they work by testing, and performed an evaluation to compare the two designs. We completed both designs successfully. The alternative design had the same structure as the baseline design except that it had additional control signals to support full bypassing; additional datapath structures, such as two muxes; and modifications to the stall and squash signals accounting for the bypassing. The only differences in datapath for alternative compared to baseline are two four-input multiplexers. Even though the alternative design includes bypassing, it includes fewer stall signals than the baseline design, requiring only a few extra transistors. Thus, we expect the cost increase of the alternative design over the baseline design to be negligible. The alternative design cycle time would likely be slightly longer than that of the baseline design due to the bypass paths potentially adding more time into the critical path. The power increase for the alternative design wouldn't be huge since there is only a modest increase in transistors for the alternative design, and the additional power usage will be offset by the slower cycle time. In the end, both designs will have similar power usage. The alternative design requires a slight increase in complexity due to the added multiplexers and bypass logic. The number of cycles executed with the alternative design was less than the baseline design when we ran five different programs: vvadd-unopt, vvadd-opt, cmult, bsearch, and mfilt. In the best-case scenario, the alternative design executed the program with 2695 fewer cycles than the baseline design (for mfilt). In the worst-case scenario, the alternative design required the same number of cycles as baseline (589 cycles for vvadd-opt program). Therefore, we can see that indeed in most programs, the alternative design minimized the number of cycles executed compared to the baseline. The CPI ratio of the baseline to the alternative design, as shown in Table 1, is at least one. In some programs like mfilt and bsearch, the CPI was larger than one. We can conclude that the alternative design is more efficient than the baseline design saving multiple cycles for programs and having lower CPI on average than that of the baseline design. The alternative design seems a better implementation of the pipelined processor compared to the baseline due to the improved performance and the minimal downsides of the slightly increased area, energy usage.

Section 2. Baseline Design

The baseline design for this lab assignment is a five-stage stalling processor that supports the TinyRV2 ISA. We divided the baseline design into two separate modules: the datapath that includes structures to process the input data in various ways depending on the control signals, and the control module, which includes signals and structures to manage the movement of data through the datapath. The pipelined processor is divided into five stages: F (Fetch Stage)- to fetch instructions and increment PC; D (Decode Stage)- to decode instructions, read register operands, and handle jumps; X (Execute Stage)- to handle arithmetic operations, address generation, and branch comparisons; M (Memory Stage) - to access data memory; W (Writeback Stage)- to write to the register file. The baseline design is illustrated in figure 2, where the blue signals display the control and status signals between the control and datapath units. The code given to us implemented the three primary instructions, add, lw, and bne; the csrr (move from the test manager); and csrw (move to the test manager) instructions. An initial framework design was given to base our design on, as shown in figure 3, and we implemented some changes in the datapath and the control unit to produce our final design. In the F stage, We modified the pc_sel_mux_F to accept four inputs, br_target_X, be_plus4_F, jal_target_D, and jalr_target_X. That change enables the execution of JAL and JALR instructions. The instructions our baseline design should implement are shown in figure 4. The other changes we implemented in datapath are adding a multiplier labeled imul in X stage, a mux in D stage called op1_sel_mux_D, two registers between the D and X stage, a +4 structure called pc_incr_X in X stage, and a mux called ex_result_sel_mux_X in X stage. The new structures we implemented are highlighted in figure 5. To implement the mul instruction, we need to add the multiplier we implemented in lab 1 in the X stage to execute the multiplication of the two register operands. The two inputs fed into the multiplier come from the op2_sel_mux_D and op1_sel_mux_D shown in Figure 2 that will be explained later in this paragraph. The op2_sel_mux_D and op1_sel_mux_D feed their outputs either to the imul or to the two registers op1_reg_X and op2_reg_X that will later feed to alu in X. We added register dmem_write_data_reg_X between the D and X stage and a wire out of it connected to the signal dmemreq_msg_data. We added a wire passing the value rs2 from the regfile to the added register. This addition was essential for the sw instruction to send the value of rs2 to mem through the dmemreq_msg_data signal. A mux ensures that the data sent to memory arrives at the correct time. To implement JAL, we need to store PC + 4 into the register rd. We added a path from the PC register into the X stage, as well as adding an additional +4 module. We needed this separate +4 module since the ALU is used to update the PC. We added ex_result_sel_mux_X to select which values to be passed to M. The mux can take three inputs: pc+4, the output from the alu, and the output from the multiplier. This is important so that we can pass pc+4 for JALR, output for arithmetic instructions like add, or, sub, and the output from mul when needed. JAL also involves passing into the PC the value of PC added to the sign-extended immediate. As such, we pass the value from pc_reg_F and the immediate into the adder pc_plus_imm_D then pass the result as the input jal_target_D to pc_reg_F. Thus, we had to add the input jal_target_D in pc_sel_mux_F. To implement JALR, we passed PC+ 4 into the register rd, similar to JAL. Then, we passed into PC the value of register rs1 added to the sign-extended immediate. To do that, we passed the value of rs1 from the regfile to op1_sel_mux_D into op1_sel_X register and passed the immediate value into op2_sel_mux_D into op2_sel_X register. Then, we added values of the imm and rs1 in the alu, passing the value from alu using a wire to the pc_sel_mux_F through the input jalr_target_X. We added in this stage the input jalr_target_X into pc_sel_mux_F so we can pass the rs1 + imm value to pc in JALR instruction. We also added the mux op1_sel_mux_D to select rs1 or PC to be passed into alu or the imul. We didn't have to add that mux previously since we didn't need to add a pc with any value.

For the control module, we added stall and squash signals to control when to stall or squash an instruction. This is important, for example, when we fetch the next two instructions after a branch instruction, but the branch is taken, as we have to squash the two instructions. In F, we generate a stall signal if the instruction in F is valid, but there is no instruction response signal sent. We stall the instruction in F if it is valid (i.e., it exists in F), and a stall was originated by instructions in F, D, X, M, or W. We squash the instruction in X if it is valid and a squash was generated by an instruction in D or X since we don't need to originate squash signals in M or W. In D, we originate a stall if the instructions in both stages are valid, the register we are writing to in stages X, M, or W is being read from in D, and the register being written to is not zero since we won't need to stall then as the value of x0 is 0. This would occur, for example, if there are two arithmetic instructions, and one writes to a register that the other reads from. Another example is when we have a load instruction writing to a register file an arithmetic instruction reads from in the second cycle. We copy the same logic for the second register rs2 ending with six originating stall signals in D. We also originate a stall if there are two multiplication signals back to back to avoid loading values into the multiplier until it is ready. We originate a squash by an instruction in D if it is valid, and there is a jump instruction in D. That is needed in case a branch is taken, and we need to squash any instructions wrongfully fetched or decoded after the branch. We stall a signal in D if it is valid, and a stall was originated by an instruction either in D, X, M, or W stage. For every stage, we stall the instruction if the current stage or any later stages originate a stall signal. Similarly, we squash an instruction in any stage if any later stages originate a squash signal. We originate a stall in M if the instruction is valid and the signal dmem_resp that sends data from memory is not valid, while the instruction type uses memory by checking that dmemreq_type_M is valid. In W, we stall if the instruction is valid and it originated the stall. This design is a good baseline for comparison since it is a fully implemented processor for the TinyRV2 ISA, but it is still fairly rudimentary. The design demonstrates many key features of a pipeline processor, such as handling data hazards, but there is still substantial room for improvement. Using a pipelined design as a baseline as compared to a single-stage processor is useful since the tools we have do not give us a precise measure of the cycle time, so we would not be able to accurately compare the performance of a single cycle design to our alternative bypassed design. Splitting the control and datapath modules make it easier to read our design, and therefore easier to debug. We could determine if any issues we had were with the datapath or the control module and then identify the error. Again, we used a system of valid/ready response and answer signals to ensure our design is able to handle inconsistencies. These signals allow our design to work independently of the time it takes to fetch instructions and data from memory, for example. This system also allows us to use multiple cycles for the X stage of multiplication operations, as the mult module does not take a consistent amount of time for any arbitrary input.

Section 3. Alternative Design

The alternative design is a five-stage bypassing processor for TinyRV2 ISA. We copied the baseline design into the alternative design and added two muxes to support the bypassing paths from the end of X, end of M, and beginning of W stages to the D stage. As shown in figure 6, we added two muxes in the Decode stage after the register file. The two muxes both take the same four inputs that are passed by the three bypassing paths, and the fourth is the value from the register file. The bypassing path from the end of X allows us to eliminate data hazards for arithmetic instructions, such as add, addi. The bypassing path from the end of the M stage minimizes the stalling needed for data hazards created by a load instruction being followed by an arithmetic instruction. The W bypass path allows us to pass values in the same cycle after the M and X stages for that instruction have been executed. We implemented modifications to the stall and squash signals accounting for the bypassing ability. For instance, we don't need to stall if we have a data hazard between two following arithmetic instructions. We have to stall for one cycle in the alternative design compared to stalling for two cycles in baseline when we have a data hazard for a load instruction followed by an arithmetic instruction. We don't include any bypassing logic in the Fetch stage, so the squash and stall signals implemented in the Fetch stage are similar to those in the baseline. In D, we add two muxes in the alternative design and feed into each of them the same four inputs: the value bypassed from X stage, M stage, W stage, or read from the register file. One reason to generate a stall signal in D is when instruction is a load followed by instruction reading the register file load writes to. To do this, we check that a valid load instruction is in the X stage and that the following instruction is dependent on the output of the load. We generate a stall signal in D if we have two instructions that will use the multiplier in the same cycle. All the bypass signals implemented ensured that the registers being written to are not 0, and the instructions at the stages involved are both valid using valid bits. For all of our bypass signals, we check that there are dependencies between the destination register of the older instruction and the source register for the younger instruction. We check that the instructions are valid and that the instructions are at a stage where the data is ready. We then set the bypass multiplexer control signals so that the correct data is used. We implemented all the bypass signals for both registers, rs1 and rs2. We implemented squash signals similar to those implemented in the baseline design, such as squashing an instruction in D if it is valid and a squash was generated by instruction in X. Similarly to our baseline design, we stall any stage if that stage or any later stages originate a stall signal. We used combinational logic using always block to select the inputs for the two bypass muxes for both registers, rs1 and rs2. For instance, if the bypass signal from X to D for register 1, rs1, is high, then we choose the input of the first bypass mux as the bypassed data from the end of X. If none of the bypass signals from X to D, M to D, or W to D is high, then we implement the default case of passing the value from regfile. Similar logic was used to select the input of the second bypass mux. In M, we originate a stall if our processor has sent a valid request to memory but has not yet received a valid response. In M, the valid signal for the next stage is high as long as there is an instruction in M and M stage is not stalling. As mentioned in the introduction, the alternative design provides better performance than the baseline design in most cases, as less stalling is required due to the bypassing. At the same time, the downsides of the slightly increased area, energy usage, and complexity are minimal. Other potential alternative designs could have included increasing the number of pipeline stages or implementing the multiplier in a way that would be less of an impedance. These designs, however, would have been challenging to implement as the base design does not lend itself to more segmentation, and separating out the multiplier would require substantial changes to the control logic. Without knowledge of the timing of each path, these designs would likely not be worthwhile.

Section 4. Testing Strategy

We tested our implementation in stages. We first ensured that every component was working individually. We used directed testing to verify that we had the expected performance on potential corner cases, such as various signed values. We next put various operations in sequence. We selected specific dependencies to ensure we were stalling or bypassing correctly. After we completed these directed tests, we ran random tests for most of the operations. This allowed us to verify our designs worked in an average case, as well as potentially illuminating any cases we had missed while implementing directed cases, though this should not be relied on. The random cases also ensured that our designs worked for a variety of values, instead of just those that we hand-picked. Finally, we implemented test cases with random delays. This ensures that our valid/ready interfaces between various components are working correctly.

We first implemented unit tests to ensure that all of our data path components were working correctly. We implemented 14 tests to check the performance of the ALU and the immediate value generator. For the immediate value generator, we created multiple tests for each immediate type, trying out positive and negative values to ensure that everything was being extended as expected. For every ALU operation, we implemented at least five different tests. We tried out different positive and negative values, ensuring that our program correctly handled values as either signed or unsigned.

We next implemented tests on the larger operations that our CPUs needed to perform. In total, we implemented 268 test functions, each of which tested many different cases. First, we ran each operation in isolation if possible. This would allow us to ensure that everything in the larger control system and datapath outside of the ALU was working as expected. Once we were satisfied with the performance of these, we implemented cases where there would be dependencies between the input registers and the output registers of other functions. This ensured that there were no issues with the stalling and bypassing. We used a wide variety of different input-output register patterns and tested cases where some registers were identical within a single instruction. We tested a variety of different numbers of NOPs between the instructions to make sure that the dependencies behaved correctly no matter what stage each instruction was in. We also identified some specific corner cases that could likely be problematic. One of these was if two multiply instructions follow each other. Due to the unique way the multiply instructions move through the pipeline, there was the potential that the data would incorrectly be loaded to or output from the multiply unit. We checked this case, as well as many other similar ones.

This method worked well for the arithmetic type operations, but additional thought was required for the control operations, such as branches and jumps. To test these, we used addi instructions to track the control flow. We would set a given bit depending on the path the program takes. This way, we could check that these operations behave correctly. We again implemented a variety of dependencies but also checked to ensure that only what we wanted to execute was actually executed.

Finally, we implemented random tests. While for certain instructions, such as a small number of the control flow instructions, we were unable to devise a way to randomly test the cases in a way that would have meaningful results, random tests played a key role in checking the arithmetic operations. We would send in random input values and ensure that the output was the expected result. We also implemented delays in our tests to ensure that our CPU was robust against any issues on the inputs.

Section 5. Evaluation

We evaluated the performance of both the baseline design and the alternative design. We observed the performance on five benchmark programs, as seen in Table 1. The first program, "vvadd-unopt," is an unoptimized version of vector-vector add. This adds the items of two identically sized vectors together elementwise. The assembly code loads the two values, adds them, and then stores them back. This creates many load-use and use-store dependencies. The Baseline design performed moderately, with a CPI of 2.22. The alternative design was able to massively reduce the number of required cycles since it would only need to stall for load-use, so the CPI was 1.34. This increase in performance, however, was not seen with the "vvadd-opt" program. This code does the same action, but it is better optimized. The code loads multiple values at the same time and then adds the corresponding pairs so that there are minimal stalls necessary. As such, there is no difference between the two designs for this benchmark. The difference between the performance on these two programs shows how knowledge about a pipelined system may be used while developing assembly code to substantially increase performance. This advantage comes at the cost of the assembly code only being optimized for a small subset of microarchitectures within a given ISA, so it would not perform as well in general.

The next benchmark program was "cmult." This program multiplies two arrays of complex numbers together elementwise. The program loads the real and imaginary components of each array entry, multiplies them together, subtracts the required cross multiplied terms, and then stores the result. The performance benefit of the alternate design was again not as substantial as found in the "vvadd-unopt" code. This is due to the fact that the multiplication instructions may stall out the whole pipeline for multiple cycles. The no bypassing can help speed this up much, so the improvements are modest.

The next benchmark program is "bsearch." This code performs a binary search of an array. Many instructions rely on the outcome of previous instructions, so there is a substantial performance improvement with the alternative design over the baseline. The alternative implementation cuts the required number of cycles down by a factor of two. This type of performance gain is also seen in the "mfilt" baseline program. This program applies a filter to a two-dimensional array. Again, there is substantial reliance on the result of previous operations, so the bypassing substantially speeds up the operation. The CPI, however, is still relatively high even for the alternative design, as there are many multiplication operations that consume many clock cycles.

Both implementations will likely require a similar amount of area to implement. The only differences in the datapath are two four-input multiplexers. The control logic for the alternative design is slightly more complicated than that of the baseline design, but it also requires fewer stall signals, so the alternate design will likely only require a small number more transistors than the baseline design. As such, we would expect both designs to cost a similar amount, as the additional logic for the alternate design is small in terms of the total logic required.

The cycle time for the alternate will likely be slightly longer than that of the baseline design. This is due to the bypass paths. This all depends, however, on which components lie on the critical path. If reading from the register file is slow, then the bypath paths may not add any additional time. This, however, is not realistic. Operations such as reading from memory are likely among the slowest possible operations. If, in the baseline design, reading from memory was the critical path, the critical path for the alternate design would be reading from memory, sending the data through the bypass muxes and the alu input selection muxes, and then into the X stage registers. If the multiplexers are quick, this increase may not be too substantial. The analysis would be the same if the critical path for the baseline design is through the ALU. The alternative design critical path would be through the X stage and through all the muxes in the D stage. Since multiplexers are generally relatively fast, especially when there are only a few inputs, the cycle time increase will likely not be too substantial to outweigh the benefits of the bypass paths in general cases. In cases such as the optimized vector-vector add, the alternative design would do nothing but slow the CPU down.

The bypassing design will likely use a bit more energy per cycle than the baseline design. This is due to the fact that there is a bit more logic, and therefore more power is required. This increase, however, will likely not be too substantial, as there is likely only a modest increase in the number of transistors required. This additional power usage per cycle will also partially be offset by the slower cycle time, so in the end, the two systems will likely have similar power usage.

The alternative design is likely a better choice in general. The two designs will require a pretty similar area to implement and use a similar amount of power to run. The primary difference is the CPI and the cycle time. The increased cycle time of the alternative design is likely small, as there are only two additional multiplexers on the probable critical path. While, for some specific applications, the bypass paths are not frequently used, so this additional cycle time would hinder the performance, for most applications the CPI improvement is such that there will likely be a substantial net decrease in execution time by the bypassed processor compared to the baseline. Out of all the benchmark programs, the ratio of total cycles of the baseline design over the total cycles of the alternate design is 1.61. Therefore those two muxes would need to slow the critical path down by a factor of 1.61 for the baseline to perform better. As such, the alternative design will perform better than the baseline in most scenarios, and the downsides from the increased area and energy usage are minimal, so the bypassed design should be preferred over the baseline.

Section 6. Role and Task Table

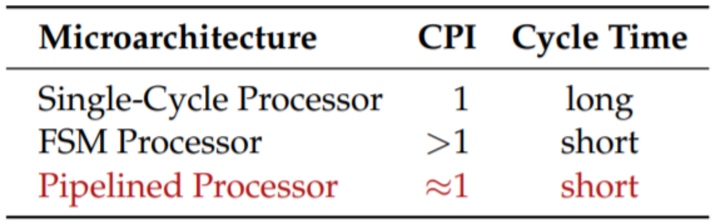
Member 1: Peter Wade	**Member 2: Lakshmi Aparna Bolla**	**Member 3: Basant Khalil**
RTL Design Engineer + Architect Adding the needed muxes and registers in the baseline design according to the dataflow path module and pseudocode. Adding code to connect these modules together to create the desired result. Providing suggestions for the RTL Verification Engineer for different possible ways to create a certain test case like providing suggestions of what values to use in testing and ensuring we are testing corner cases. Fixing bugs in the test cases that were pushed to verilog where he used the waveform to debug some test cases like the bne. Contributing to the lab report through computing to different sections, such as evaluation and testing. Also, creating the table comparing between the performance for the alternative design versus the baseline design. Explaining to a team member how to fix some errors that appeared when writing the bypass logic in the control module for the alternative design. Creating a block diagram that shows the datapath for the alternative design in figure 6.	RTL Design Engineer Helping with debugging some bugs in the main baseline code. For instance, reminding the lab members of how to view the wave to help with debugging of the errors showing up. Helping write the logic for the control module in the alternative design through mentioning the bypass logic needed in many stages, like X and M, and how to modify the squash and stall logic implemented to account for bypassing in the alternative design. Helping create test cases for instructions like the jump instructions and helping with debugging for those cases. Providing suggestions to other team members along the way of fixing the errors showing up. Helping with writing the code for the baseline design and providing suggestions to help with debugging the design when we faced some errors. Contributing to the lab report through contributing to sections like referenced figures, introduction and alternative design.	RTL Verification Engineer Writing Test cases, such as sw test cases, and debugging along with the other members the errors that showed up until the code for the baseline design passed the test cases. Helping with coding the squash and stall logic in the control module of the baseline design and providing some suggestions to other team members. Providing some suggestions to fix some of the errors that existed in the code, such as helping with fixing some of the bypass logic errors associated with the alternative design. Contributing to the lab report through contributing to sections, such as introduction, baseline design and alternative design sections. Writing some logic for the modified stall and squash signals in the control module for alternative design. Coming up with the diagram that highlights the extra data path structures shown in figure 5 needed in the baseline design to end up with desired design in figure 2 starting with the code given to us implementing the initial baseline design in figure 3.

Section 7. Referenced Figures

Table 1: The cycle counts for various evaluation programs.

	Number of Cycles		Number of Instructions	CPI		CPI Ratio (Base/Alt)
	Base	Alt		Base	Alt	

vvadd-unopt	2014	1214	907	2.22	1.34	1.66
vvadd-opt	589	589	532	1.11	1.11	1
cmult	4126	3826	1707	2.42	2.24	1.08
bsearch	4537	2116	1527	2.97	1.39	2.14
mfilt	5179	2484	1350	3.84	1.84	2.09



Microarchitecture	CPI	Cycle Time
Single-Cycle Processor	1	long
FSM Processor	>1	short
Pipelined Processor	≈1	short

Figure 1: showing the estimated CPI (Cycles Per Instruction) and Cycle Time for 3 common types of microarchitectures: Single-Cycle Processor, FSM Processor, and Pipelined Processor

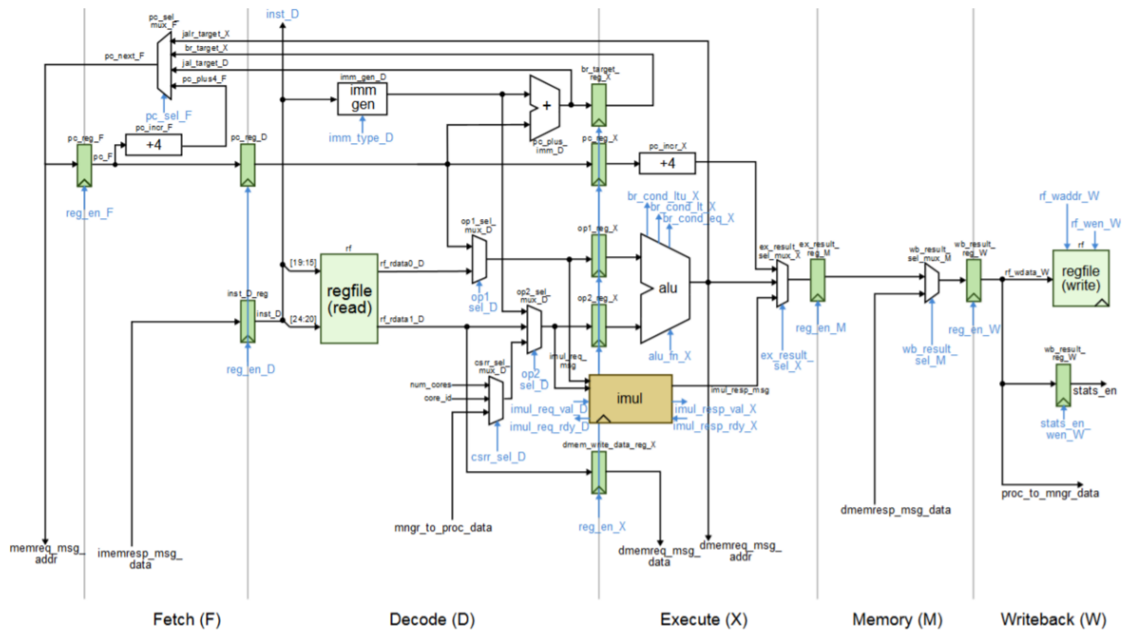


Figure 2: Baseline Design: Five-Stage Stalling Processor Datapath

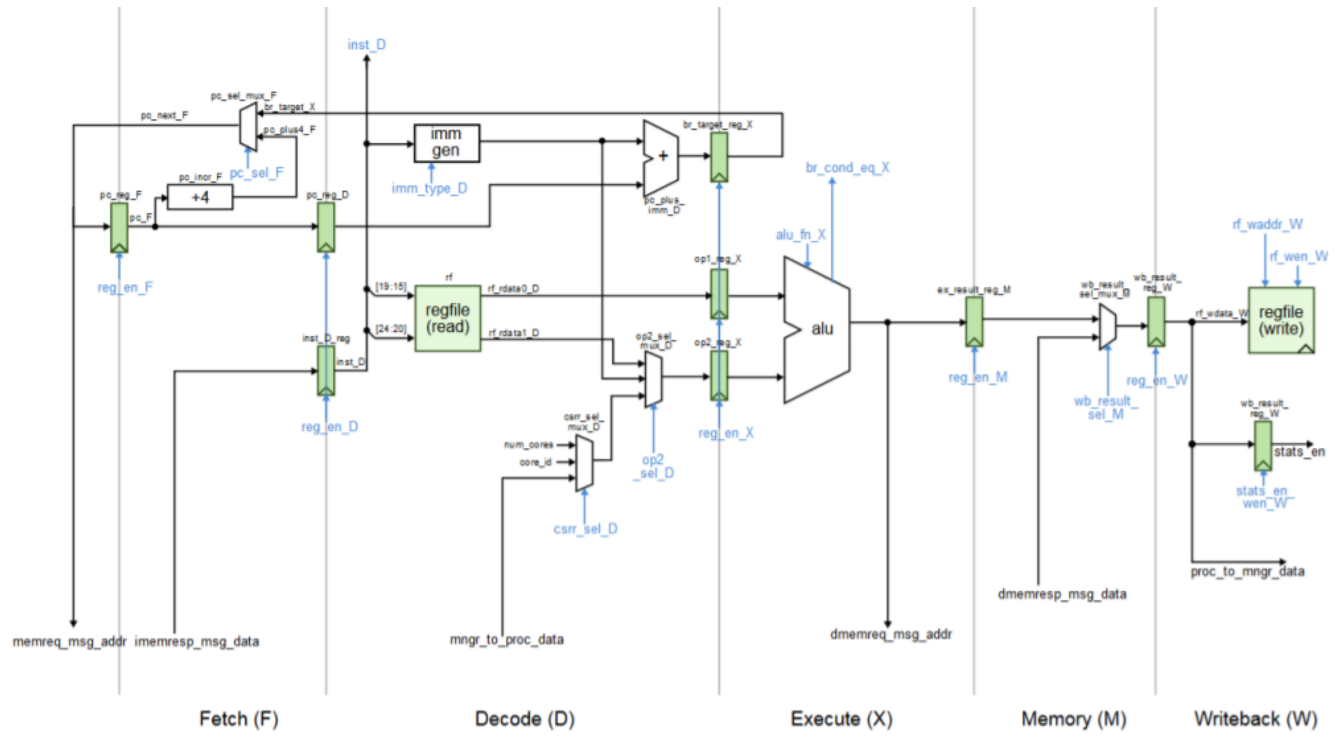


Figure 3: Initial Baseline Design Provided To Students

- CSR : `csrr, csrw`
- Reg-Reg : `add, sub, mul, and, or, xor, slt, sltu, sra, srl, sll`
- Reg-Imm : `addi, ori, andi, xori, slti, sltiu, srai, srli, slli, lui, auipc`
- Memory : `lw, sw`
- Jump : `jal, jalr`
- Branch : `bne, beq, blt, bltu, bge, bgeu`

Figure 4: The Instructions Our Design Should be Able to Implement

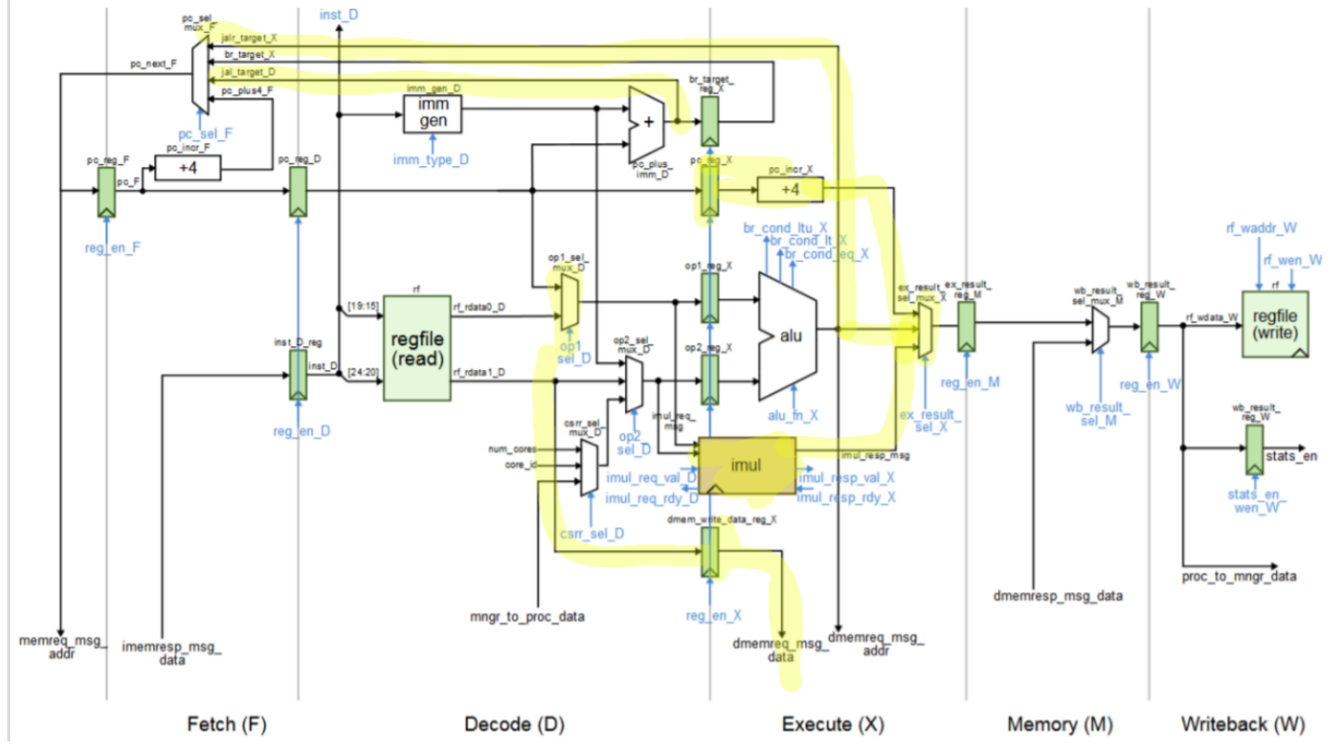


Figure 5: Highling the New Datapath Structures We Added In The Baseline Design Given to Us to get the Baseline Design in Figure 2

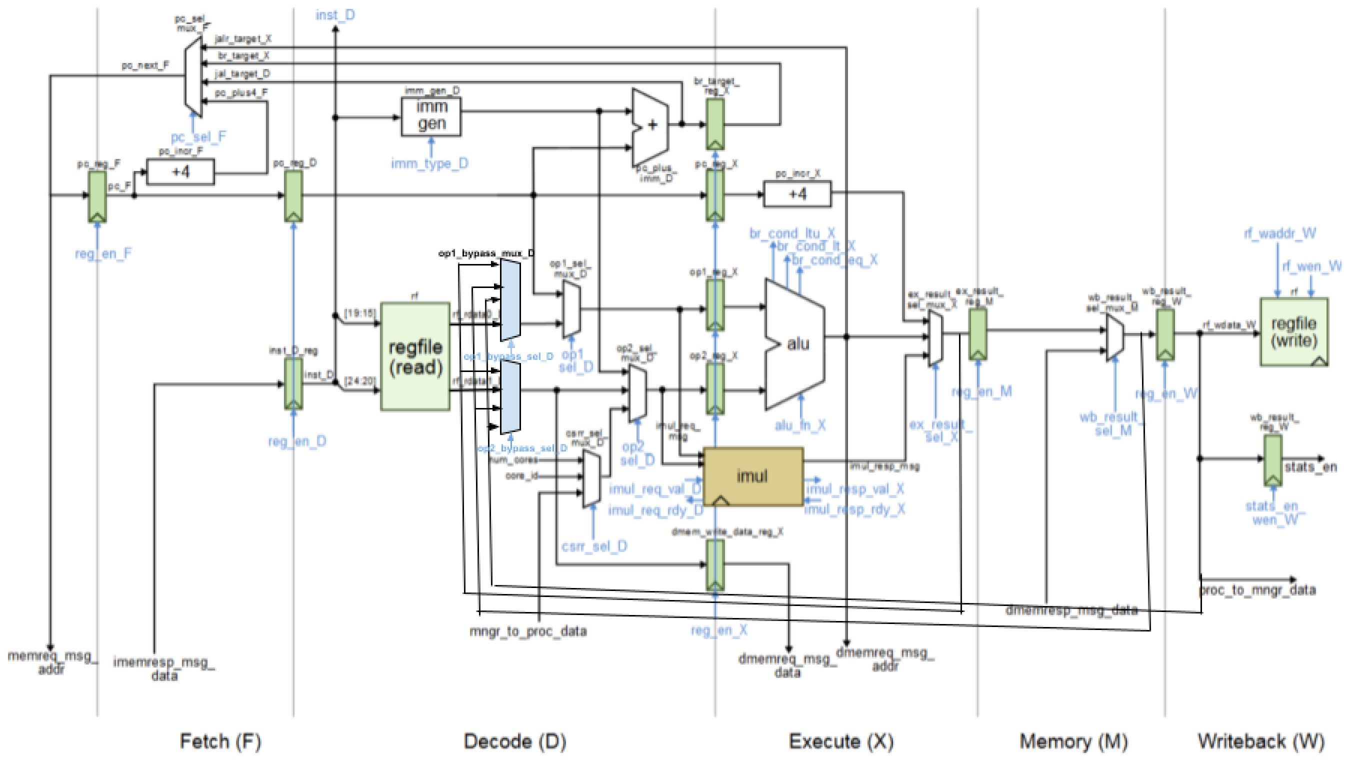


Figure 6: Alternative Design Diagram as a Bypassing design